Amendments to the Specification

Please replace the paragraph beginning at page 1, line 1 (i.e., the title), with the following rewritten paragraph:

--NETWORK-INDEPENDENT PROFILING OF APPLICATIONS FOR AUTOMATIC PARTITIONING AND DISTRIBUTION IN A DISTRIBUTED COMPUTING ENVIRONMENT--

Please replace the paragraph beginning at page 8, line 27, with the following rewritten paragraph:

--Figure 7 is a flow chart showing the scenario-based profiling of an application to generate a description of the run-time behavior of the application according to the illustrated embodiment of the present invention.--

Please replace the paragraph beginning at page 10, line 14, with the following rewritten paragraph:

--Figures 1 and 2 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the illustrated ADPS can be implemented. While the present invention is described in the general context of computer-executable instructions that run on computers, those skilled in the art will recognize that the present invention can be implemented as a combination of program modules, or in combination with other program modules. Generally, program modules include routines, programs, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The present invention can be implemented as a distributed application, one including program modules located on different computers in a distributed computing environment.--

Please replace the paragraph beginning at page 20, line 14 with the following rewritten paragraph:

--Referring again to Figure 4, cross-machine communication occurs transparently through [[and]] an interface proxy 110 and stub 130, which are generated by software such as the MIDL compiler. The proxy 110 and stub 130 include information necessary to parse and type function arguments passed between the client 100 and the component 140. For example, this information can be generated from an Interface Description Language (IDL) description of the interface of the

component 140 that is accessed by the client 100. The proxy 110 and stub 130 can provide security for communication between the client 100 and the component 140. A client 100 communicates with the proxy 110 as if the proxy 110 were the instantiated component 140. The component 140 communicates with the stub 130 as if the stub 130 were the requesting client 100. The proxy 110 marshals function arguments passed from the client into one or more packets that can be transported between address spaces or between machines. Data for the function arguments is stored in a data representation understood by both the proxy 110 and the stub 130. In DCOM, the proxy 110 and stub 130 copy pointer-rich data structures using deep-copy semantics. The proxy 110 and stub 130 typically include a protocol stack and protocol information for remote communication, for example, the DCOM network protocol, which is a superset of the Open Group's Distributed Computing Environment Remote Procedure Call (DCE RPC) protocol. The one or more serialized packets are sent over the network 120 to the destination machine. The stub unmarshals the one or more packets into function arguments, and passes the arguments to the component 140. In theory, proxies and stubs come in pairs—the first for marshaling and the second for unmarshaling. In practice, COM combines code for the proxy and stub for a specific interface into a single reusable binary.--

Please replace the paragraph beginning at page 31, line 24, with the following rewritten paragraph:

--Dynamic analysis provides insight into an application's run-time behavior. The word "dynamic," as it is used here, refers to the use of run-time analysis as opposed to static analysis to gather data about the application. Major drawbacks of dynamic analysis are the difficulty of instrumenting an existing application and the potential perturbation of application execution by the instrumentation. Techniques such as sampling or profiling reduce the cost of instrumentation. In sampling, from a limited set of application executions, a generalized model of application behavior is extrapolated. Sampling is only statistically accurate. In profiling, an application is executed in a series of expected situations. Profiling requires that profile scenarios accurately represent the day-to-day usage of the application. A scenario is a set of conditions and inputs under which an application is run. In the COIGN system, scenario-based profiling can be used to estimate an application's run-time behavior.--

Please replace the paragraph beginning at page 42, line 3, with the following rewritten paragraph:

--Each edge in the commodity-flow graph effectively represents the cost in time of distributing that edge. Because the common currency of graph edges is time, other time-based factors that affect distribution choice can be mapped readily onto the same MIN-CUT problem with communication costs. A good example is the problem of deciding where to place application units when client and server have different speed processors. For this case, two additional edges are attached to each application ~~units~~ unit. An edge from the application unit to the source *s* has a weight equal to the execution time of the application unit on the server. A second edge from the application unit to the sink has a weight equal to the execution time of the application unit on the client.--

Please replace the paragraph beginning at page 43, line 3, with the following rewritten paragraph:

--In the illustrated ADPS, accurate values of latency and bandwidth for a particular network [[ca]] can be quickly estimated using a small number of samples, enabling adaptation to changes in network topology including changes in the relative costs of bandwidth, latency, and machine resources.--

Please replace the paragraph beginning at page 72, line 21, with the following rewritten paragraph:

--According to one method of static binding of the COIGN RTE into an application, the application binary is modified to add the RTE DLL to the list of imported DLLs. Static binding insures that the RTE executes with the application. Referring to Figure 15, an application binary 600 in a common object file format ("COFF") includes a header section 610, a text section 616, a data section 620, a list of imports 630, and a list of exports 640. The header section 610 includes pointers 611 – 614 to other sections of the application binary 600. The text section 616 describes the application. The data section 620 includes binary data for the application. Within the binary data, function calls to functions provided by other DLLs are represented as address offsets from the pointer [[612]] 613 in the COFF header 610 to the imports section 630. The list of imports includes two parallel tables. The first table, the master table 632, contains string descriptions of other libraries and functions that must be loaded for the application to work, for example, necessary DLLs.

The second table, the bound table 634, is identical to the master table before binding. After binding, the bound table contains corresponding addresses for bound functions in the application image in address space. Function calls in the data section 620 are directly represented as offsets in the bound table. For this reason, the ordering of the bound table should not be changed during linking. The exports list 640 includes functions that the application binary 600 exports for use by other programs.--

Please replace the paragraph beginning at page 73, line 12, with the following rewritten paragraph:

--To statically bind the COIGN RTE into an application, COIGN uses binary rewriting to include the COIGN RTE in the list of imports 630. To load the rest of the COIGN runtime DLLs before any of the other DLLs are loaded, and to modify COM instantiation APIs at the beginning of application execution, the COIGN RTE DLL is inserted at the beginning of the master table 632 in the list of imports 630. Because the application is in binary form, merely inserting the COM RTE DLL into the master table of the list of imports is not possible without replacing the first entry on the master table 632 (assuming the first entry reference had the same length), or corrupting the binary file. For this reason, a new imports section 650 is created. Into the master table [[652]] 654 of the new imports section 650, the binary rewriter inserts an entry to load the COIGN RTE DLL, and appends the old master table 632. A dummy entry for the COIGN RTE DLL is added to the bound table [[654]] 652 of the new imports section 650 to make it the same size as the master table, but the dummy entry is never called. The bound table is otherwise not modified, so the references within the COFF binary data to spots within the bound table are not corrupted. The header section 610 of the application points 618 to the new imports section 650 instead of the old imports section 630. At load time, the libraries listed in the new master table 650 are loaded. Addresses are loaded into the new bound table 654. Function calls from the data 620 of the COFF continue to point successfully to offsets in a bound table. In this way, the COIGN RTE DLL is flexibly included in the list of imports without corrupting the application binary. The application is thereby instrumented with COIGN RTE, and the package of other COIGN modules loaded by the COIGN RTE according to its configuration record.--

Please replace the paragraph beginning at page 74, line 6, with the following rewritten paragraph:

--To dynamically bind the COIGN RTE DLL into an application without modifying the application binary, a technique known as DLL injection can be used. Using an application loader, the RTE DLL is forcefully injected into the application's address space. Inserting a code fragment into an application's address space is relatively easy. With sufficient operating-system permissions, the Windows NT virtual memory system supports calls to allocate and ~~modifying~~ modify memory in another process. After the application loader inserts a code fragment into the application's address space, it causes the application to execute the fragment using one of several methods. The code fragment uses the LoadLibrary function to dynamically load the RTE DLL.--

Please replace the paragraph beginning at page 76, line 4, with the following rewritten paragraph:

--As shown in Figure 16, an application binary 600 in common object file format ("COFF") includes a header 610, text [[619]] 616, data 620, an imports list 630, and an exports list 640. The imports section 630 includes master 632 and bound 634 tables. To reversibly link a library to the application binary 600, a header 660 is appended to the application binary 600. In COIGN, the appended header 660 is called a COIGN header. The original COFF header 610 is copied to the appended header for storage.--

Please replace the paragraph beginning at page 77, line 3, with the following rewritten paragraph:

--To re-link the application binary, the original COFF header 610 is restored from the appended header 660. The appended header 660, new imports section 670, and any appended data 680 are discarded. Because the original COFF header 610 contained a pointer [[614]] 613 to the original imports section 630, the application binary 600 is restored. At this point, the process can be repeated using the original application binary, or using a second library instead of the first library. Alternatively, the first entry 673 in the master table 672 of the new imports section 670 can be overwritten with a binary rewriter to include the second library instead of the first, and the application re-binded.--

Please replace the paragraph beginning at page 82, line 25, with the following rewritten paragraph:

--Another alternative is to acquire static interface metadata from the COM type libraries. COM type libraries allow access to COM components from interpreters for scripting languages, such as JavaScript or Visual Basic. While compact and readily accessible, type libraries are incomplete. The metadata in type libraries does not ~~identity~~ identify whether function parameters are input or output parameters. In addition, the metadata in type libraries does not contain sufficient information to determine the size of dynamic array parameters.--

Please replace the paragraph beginning at page 98, line 6 (i.e., the abstract), with the following rewritten paragraph:

--An instrumentation system profiles an application using structural metadata description of the application. Units (such as COM objects) of the application have strongly-typed, binary-standard interfaces, and are profiled, for example, using an executable file and DLLs for the application. ~~without access or reference to application source code.~~ A structural metadata description of the application includes compiled, interface-level type information used to identify and measure interaction between units of the application. For example, the type information is produced by analyzing IDL information. Profiling ~~of the application~~ results in an application profile that includes description of the static relationships and/or dynamic interactions between units of the application. The application profile is combined with a network profile that describes a distributed computing environment. Analysis of the result yields a distribution plan ~~for units of the application in a distributed computing environment~~, which, for example, reduces costs associated with communication between the units. During execution, units of the application are distributed through the distributed computing environment according to the distribution plan.--